TP1: Where are the Panzers?

Victor Pires Diniz

May 7, 2015

Algoritmos e Estruturas de Dados III - 1º Semestre de 2015

1 Introduction

String processing and external sorting are important classes of problems in computer science. This work combines both of them by addressing a fictitious scenario from World War II in which messages from the Axis have been intercepted. A program must be written to, then, parse these messages, in order to clear any noise and figure out where their troops will be deployed for attacks and how many will be.

These messages are all obtained in a continuous stream of data, and the program must, therefore, identify where given patterns show up. This would be a great opportunity to use regular expressions, but C's standard library features no such procedures, sadly. Implementing a regex library would be overkill and likely much more trouble than it's worth. A good, simpler alternative to that would be to implement a string-matching automaton to parse these messages, which is what the parsing module of this project does, essentially.

Then, the Alliance must assess the priorities between each attack to establish a strategy and minimize losses. That is done by sorting the attacks, prioritizing the ones with the most tanks and, if there's a tie, the ones that are closer to the headquarters. If there's still attacks with the same priority, then they should be dealt with in the order they appeared originally. This last detail creates an issue: not only must the program be able to sort these attacks, it must do so stably.

Finally, in order to match the scenario, the program must be able to run in an environment with limited primary memory. Accordingly, it should be able to run and sort data in the hard drive, since the attacks could easily exceed the memory that's left of what the program's other variables and binary data need. Ways to solve this will be described in the next section.

2 Getting it done

The problem to be handled here, as stated before, has two distinct parts: parsing and sorting. Both of them deal with attacks, which are defined in the program as a simple type which contains three integer fields: the number of tanks (or panzers) as well as the coordinates of the attack.

2.1 Parsing

A simple automaton can be used for matching a string with given characteristics. The way it does it is by fetching each character of the string sequentially and moving between states. Since the problem states that there may be noise between the valid characters read, unexpected characters should not reset the machine, but keep it in the same state.

In this case, the pattern to be matched can be split in two: the first section is made up of 'p', followed by any number of 'o's larger than zero, 'i', 'n' and, finally, 't'. The number of 'o's read denotes the number of tanks in the attack. Any characters that don't match the expected inputs for a given state are considered noise and ignored. After 't', the program expects to read '(', which prompts the second phase of the automaton.

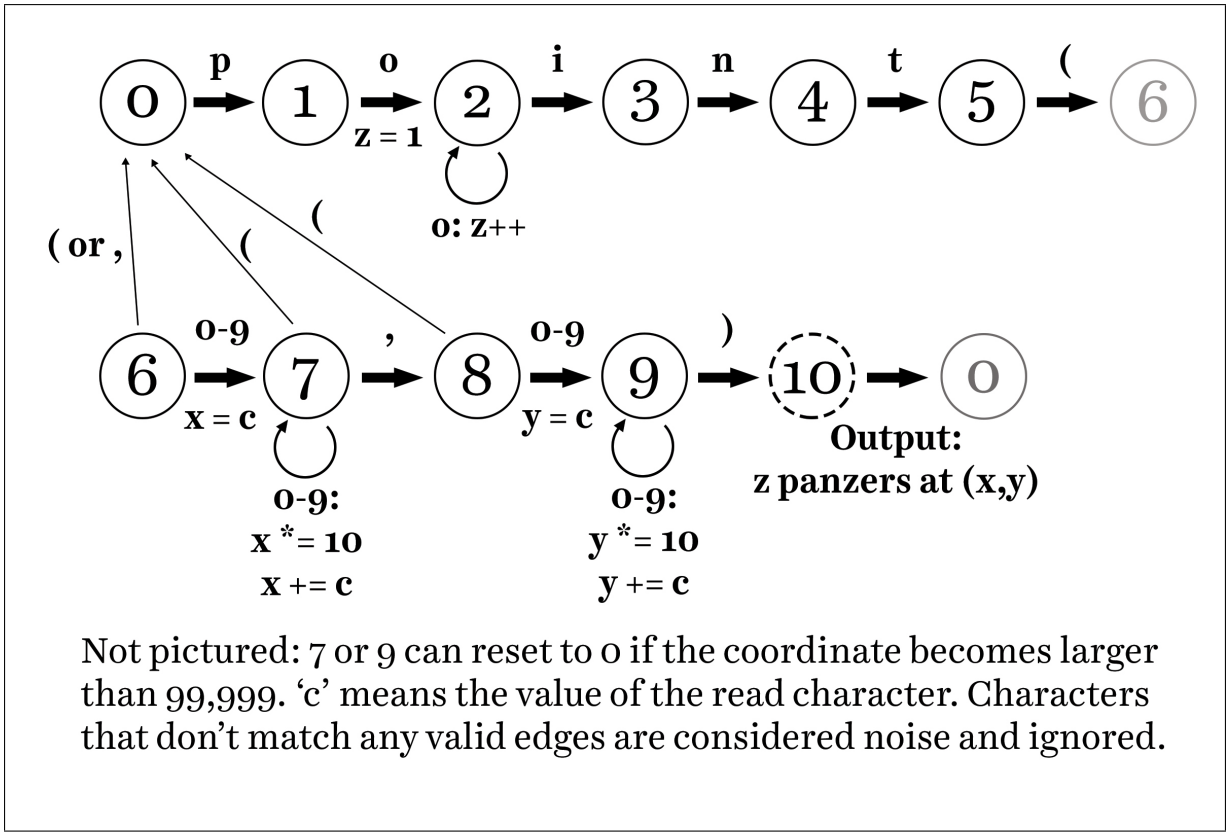

Figure 1: State diagram of the parsing automaton.

In the second phase, the automaton expects a number from 0 to 99999, a comma (',') and another number from 0 to 99999, followed by ')' in the end. These values mean the 'x' and 'y' coordinates of the attack. However, many inputs are considered invalid. The following characters under certain circumstances can make the finite state machine reset to state '0', back to when it expected to read the first 'p':

- A second '(';
- A comma before reading any valid number;
- A second comma;
- A digit which makes the current coordinate exceed the maximum accepted value.

Any other character is considered noise and is promptly ignored. When the automaton successfully reaches the final state, it prints the attack to an output file. Then, all that's left is to sort the file and print it back.

2.2 Heap

The next module, which deals with sorting, uses a heap to perform the sorting operation. The heap is a kind of binary tree in which a given node's children are smaller than itself (in the case of a max-heap). It is also always balanced, and often implemented via an array. There's no requirements for which child is larger or smaller, and, because of this, the heap is not suitable for search operations. However, its main property makes extracting the largest element very straight-forward.

Inserting an element to the heap can be done by inserting it to the first free element in the array (after all existing nodes) and enforcing the heap's main property locally: if the newly inserted element is larger than its parent, they should be swapped. Then, the same check needs to be made to the new parent node until the child node is actually smaller than the parent node or it becomes the new root, in case it's actually the largest element currently in the heap.

Removal of the largest element is also rather simple: first, the element in the root is removed and its value is saved to be returned. Then, all that's left is to fix the gap left. To do it, the last element of the heap is moved to the root, and, then, if it's smaller than its largest child, these two nodes are swapped. This operation is repeated, in similar fashion to the insertion checks, until the node is larger than both of its children or until it becomes a leaf node.

An issue with the heap is that its operations are not stable: it may swap two equivalent nodes because of the way insertion and removal work. To fix this issue, all data inserted in heaps in this project is inserted together with a wrapper, which contains an integer field that tracks the order of insertion in the heap. The comparison function used to determine the order of the heap uses this as its last priority factor. This guarantees stable heap operations.

2.3 Sorting

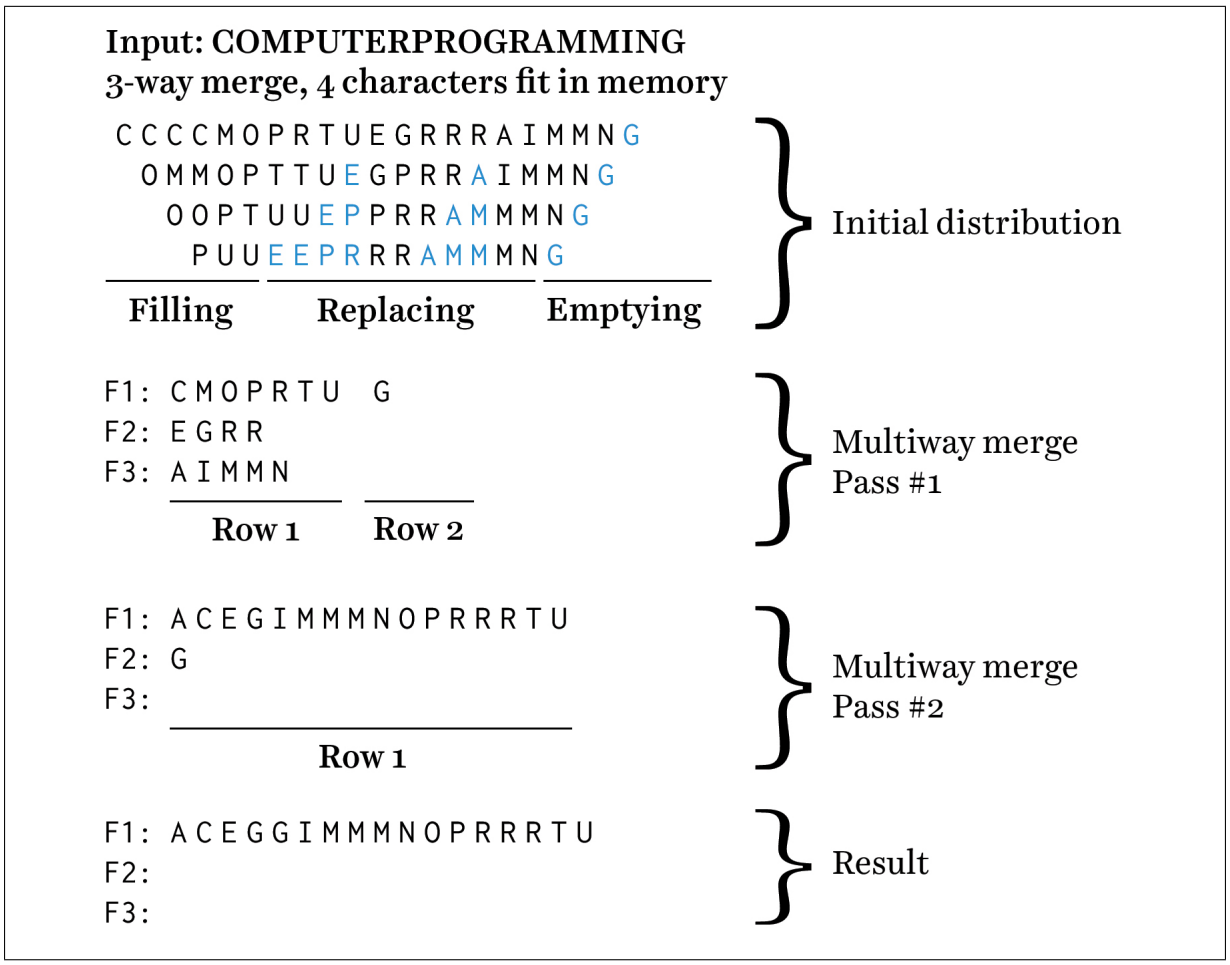

Figure 2: Diagram showing both phases of the replacement selection algorithm: the initial distribution and the subsequent multiway merges. The blue entries in the initial distribution are flagged and belong to the next block. They become black when the block switch occurs.

There are many different ways to sort a file that doesn't fit entirely in the amount of available RAM. The most popular solution is the external merge sort, which involves loading the maximum amount of elements possible to primary memory, sorting them, saving these blocks in temporary files and, then, performing an N-way merge. The sorting algorithm used in this project, which is presented in the following paragraphs, is an improvement to the regular external merge sort which uses a heap to perform both the internal sorting and the N-way merge. **Replacement selection** is a variation of the selection sort which uses an auxiliary heap to determine the smallest element, in a tournament sort-style selection.

2.3.1 Initial distribution

First of all, a heap of attacks is created, defined to be as long as possible within the memory constraints, and filled with elements from the input file until it is full. Then, in order to read the subsequent elements, the smallest (this explanation assumes an ascending sort) element is removed from the heap and written to a temporary file which will contain a sorted block of elements. The newly read element is added to the heap, but in case it is smaller than the element which was just added, it can not be inserted into the same block. Therefore, an additional field is assigned to each element added to the heap: a flag that denotes to which block it belongs. If the element that has just been read shouldn't be allowed into the current block, then its flag field should be set to the identifier of the next block. It ought to be set to the current block's ID otherwise. When all elements in the heap belong to the next block, it becomes the current block, and the ensuing removed elements will be printed to it instead. This process continues until the input file reaches its end. Then, the heap should be simply emptied.

Algorithm 1: Initial distribution pseudocode

input : An input file that contains a list of attacks (input), an array of output files of size $\#ways$ (outputs) and the size of the available memory (m).
output: Sorted blocks of attacks in the output files.

h = StableHeap(length m / (size of attack + size of flag);
a = input.ReadNext();
while $a \neq nil$ *and* h *not full* **do**
    h.Insert(a, flag $\leftarrow 0$);
    $a \leftarrow$ input.ReadNext();
end
while $a \neq nil$ **do**
    $w \leftarrow h$.pop() // remove maximum priority element from the heap;
    outputs[$w.flag \pmod{ways}$].Write(w);
    if a *has higher priority than* b **then**
        h.Insert(a, flag $\leftarrow w.flag + 1$);
    else
        h.Insert(a, flag $\leftarrow w.flag$);
    end
    $a \leftarrow$ input.ReadNext();
end
while h *not empty* **do**
    $w \leftarrow h$.Pop();
    outputs[$w.flag \pmod{ways}$].Write(w);
end
return $w.flag + 1$ // number of sorted blocks written;

This initial split method is particularly efficient because the blocks created in the first step of the procedure are, on average, twice as long as the ones created in a multiway merge, which are as large as the available memory (m). This is due to the fact that there are, initially, m entries in the heap. When a new element is read, if it is random, it should have equal chance of being larger or smaller than the element which was just removed. By itself, this guarantees that, on average, $1.5m$ elements will be output to a block. However, the extra $0.5m$ elements will also, on average, add more $0.25m$ entries to the current block. Clearly, this is the sum of an infinite geometric progression with common ratio $\frac{1}{2}$ and initial value m, which equates to $2m$.

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{1}{1-\frac{1}{2}} = 2 \tag{2.1}$$

This is advantageous because the time required for performing an external merge sort is predominantly defined by the number of runs generated by this initial distribution, which is inversely proportional to the number of blocks. Larger blocks means fewer blocks, which makes replacement selection an attractive technique. The equation below denotes the expected *average* number of runs required to finish the multiway merge, in which n means the number of entries in the unsorted file and m the amount of elements that fit in primary memory:

$$\log_{\#ways} \#blocks = \log_{\#ways} \left\lceil \frac{n}{2m} \right\rceil \tag{2.2}$$

Blocks are written to files in sequence: the first block to the first file, the second block to the second file etc. In case the number of blocks is larger than the number of ways of the multiway merge, the program should simply wrap back to the first file. A block with a given id should, therefore, belong to file $id \pmod{\#ways}$. Also, due to the fact that these files will be read from and written to, there should be two sets of temporary files available to the procedure, each one with as many files as the number of ways.

2.3.2 Multiway merge

Algorithm 2: Multiway merge pseudocode

input : An array of input files of size $\#ways$ (inputs), an array of output files of size $\#ways$ (outputs).
output: Sorted blocks of attacks in the output files.

```
h = StableHeap(length m / (size of attack + size of flag);
for i ← 0 to (#ways − 1) do
    a = inputs[i].ReadNext();
    h.Insert(a, flag ← i);
end
next = StableHeap(length m / (size of attack + size of flag);
current_file ← 0;
while h not empty do
    while h not empty do
        w = h.Pop();
        a = inputs[w.flag (mod ways)].ReadNext();
        if a.flag = w.flag then
            h.Insert(a, flag ← a.flag);
        else
            next.Insert(a, flag ← a.flag);
        end
        w.flag ← current_file;
        outputs[w.flag (mod ways)].Write(w);
    end
    Swap(h, next);
    current_file++;
end
return current_file;
```

The second part of the replacement selection method uses, once again, a heap, but this time its length is equal to the number of ways. This step is basically an N-way merge, but using a heap to optimize the operation for selecting the smallest element that hasn't been added yet. The procedure is very simple: for each row of blocks (group containing a single block of sorted entries from each file) from the initial operation, the first element of each block is added to the heap. Then, the smallest element is removed from the heap, printed to the first output file and another element from the same block as the one which was removed is added to the heap. This keeps on going until all the blocks from this row are emptied. After that, the same scheme is repeated for the next block row, with the entries being written to the next available output file this time. Once all block rows are finished, the number of existing sorted blocks in the output files should be $\lceil \#ways \rceil$ times smaller than there were before. Thus, the procedure can simply be repeated, with the output files being used as input, until there is only a single sorted block left. The only thing left is to copy this block back to the original file.

2.3.3 Choosing the number of ways

Little has been mentioned about how the number of ways is actually chosen for the sorting function. The number of ways can influence the amount of passes necessary to sort a file significantly, being the base of the logarithm in equation 2.2. There are few actual restrictions to $\#ways$.

First of all, $\#ways$ must be smaller than the amount of attacks that fit in memory, because otherwise the heap won't be able to contain an element from each file at once. It also needs to be greater than 1, because a 1-way merge makes absolutely no sense. There's also a restriction enforced by most operating systems for the maximum number of files open at once by a process. This program has at most $2 * \#ways$ files open ($\#ways$ input files, $\#ways$ output files). Other than that, it can be chosen freely. Theoretically, $\#ways$ should be maximized in order to optimize the amount of runs of the replacement selection sort, but the fact that the amount of runs is determined by a ceil function makes this much more effort than it's actually worth. Having too many files open can be detrimental to performance, and a higher number of ways only makes any difference at all if the number of runs manages to go below the threshold between two integers. Also, having a dynamic number of ways would make it necessary to make plenty of dynamic allocations in the program, which could be a huge hit to the program's performance.

All in all, the final choice was made to sensibly pick a value that made sense and would be enough for most situations and hard code it. This value is 17.

3 Assymptotic complexity analysis

Each input file contains two relevant variables for assymptotic complexity analysis: the amount of memory available (m) and the length of the input plain text (n). Also, due to the fact that external memory operations are much slower than primary memory operations, the assymptotic time complexity analysis will also be performed in regard to file input and output. If not specified, the assymptotic complexity is valid for both file and primary memory operations. If a procedure hasn't been mentioned below, it has $O(1)$ assymptotic time and space complexities, and is likely very simple, such as the comparison functions utilized in this project.

3.1 Parsing

3.1.1 parseData

This function is the main function in the parsing module. It simply obtains each character from the standard input and goes through a chain of conditionals, printing coordinates to the output file if it reaches the final state. Its time complexity is $O(n)$ concerning both its file operations and regular operations.

3.1.2 unpackData

This procedure is very straight-forward: it simply reads data from a file linearly and prints it to standard output. It is, therefore, $O(n)$.

3.2 Heap

Because it is a generic data structure, the heap will be analyzed regardless of the context of this program. It will be contextualized when used in the sorting operations. In this subsection, n means the number of elements currently in the heap and m refers to the maximum amount of elements that fit in the heap.

3.2.1 hInit

This procedure simply initializes the heap. It has $O(1)$ time complexity, but its space complexity is $O(m)$, since it allocates the entire heap.

3.2.2 hInsert

Inserting an element in the heap is $O(\log n)$, because, worst case scenario, the new element will be swapped all the way to the top of the heap, which would take $\log n$ swaps.

3.2.3 hPop

Removing an element from the heap also is $O(\log n)$, for the same reason as *hInsert*. The last element in the heap may have to be swapped down from the root until it becomes a leaf node.

3.3 Sorting

3.3.1 initialDistribution

The initial distribution procedure reads the file that contains the attacks parsed from the original input file, distributing the entries in sorted blocks. It does a single pass through its input file. Because of that, its time complexity is $O(n)$. Also, as far as space complexity goes, is has only a bunch of static variables and a single dynamically allocated heap with size proportional to the available memory, which has no relation to the input's size. Thus, its space complexity is $O(m)$ concerning primary memory. The function reads and writes to different file sets, though, and, because of that, $O(n)$ space is required in secondary memory.

3.3.2 multiwayMerge

This procedure performs a single pass through the input files with sorted blocks, merging them into longer blocks. Because it reads each attack only once, this function has linear time complexity concerning file and primary memory operations. Its space complexity is $O(1)$ in primary memory, on the grounds that the only dynamically allocated entity are the two heaps used, each one with size $\#ways$. However, similarly to the initialDistribution, $O(n)$ space is required in the disk in order to write the merged blocks.

3.3.3 saveToAddr

This function is extremely similar to unpackData, in the sense that it simply reads each entry in the final sorted block once and prints it back to the original file. It has, therefore, linear time complexity.

3.3.4 sortAttacks

The main function of the sorting module calls initialDistribution once, multiwayMerge $\log_{\#ways}\lceil\frac{n}{2m}\rceil$ times and saveToAddr one time. All of these functions are $O(n)$, which results in $O(n\log\frac{n}{m})$ for the entire procedure. The space complexity is $O(m)$ regarding primary memory, but at any given moment, enough secondary memory space for a complete copy of the input is necessary. This happens because the initial distribution and the multiway merge read and write to different sets of files, which need to be, thereupon, available to be read in the next step. This implies in $O(n)$ space complexity with respect to secondary memory.

3.4 Main

The program's primary function is extremely simple: it calls parseData, sorts the attacks with sortAttacks and prints the sorted attacks back to standard output with unpackData. $O(n)+O(n\log n)+O(n)=O(n\log\frac{n}{m})$. Its space complexity is $O(m)$, since the maximum usage of memory happens during the initial distribution, when a heap with size m is allocated.

4 Experimental analysis

For this experimental analysis section, two benchmarks will be performed, analyzing the two most relevant factors to the program's performance: the length of the input and the amount of memory available to the program. All input files were generated automatically by a test generator coded in Python, whose source code is available at the end of this documentation. The computer used to run the test has the following specifications:

- OS: Arch Linux x64;
- CPU: Intel Core i7 2600k @ 3.40GHz;
- RAM: 4 x 4GB DDR3 1.333MHz (16GB);
- SSD: 60GB Kingston SKC300S37A60G;

The actual time measurement was done through Linux command *time*. These tests were performed on the 7th of May, 2015.

4.1 Test 1: Input length

For this test batch, the program has been fed input from six different input files with varying input lengths, ten times each. All of these files had memory usage limited to 30MB. The results are available in the chart below:

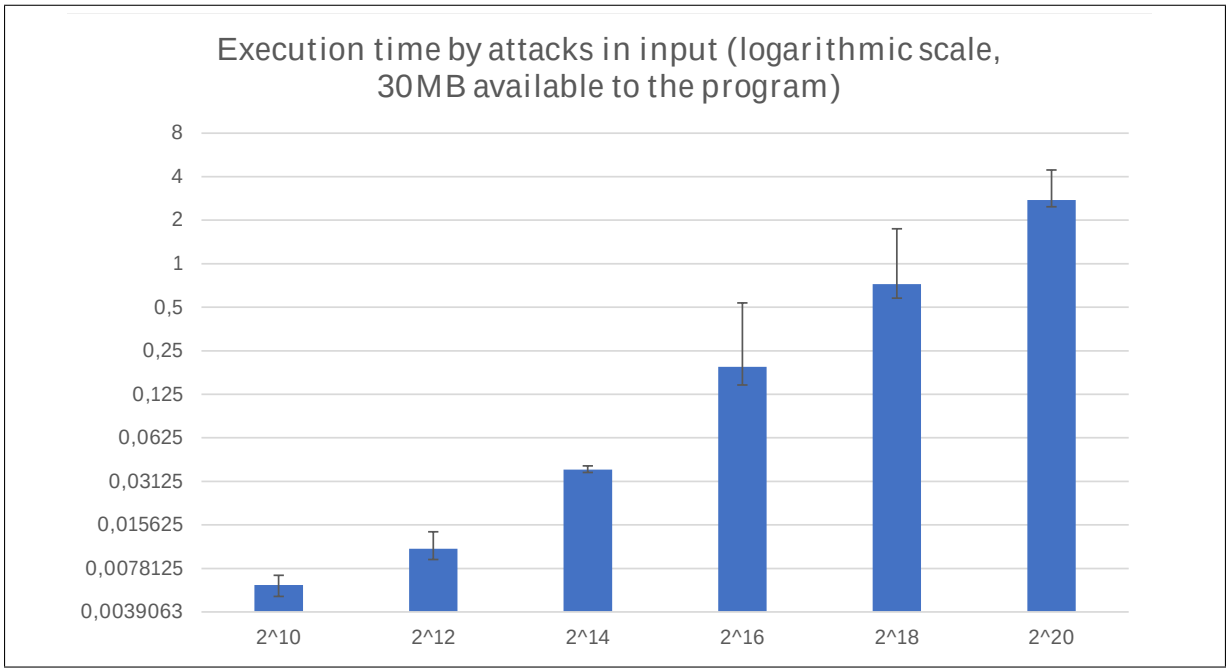

Figure 3: Chart of the execution time of the tests (in seconds) by the length of the inputs (number of attacks) in logarithmic scale (base 2).

This test behaved as expected. In the assymptotic complexity analysis section, it has been concluded that the entire program has $O(n \log n)$ time complexity, if the amount of memory available is constant. The chart seems linear in logarithmic scale, which is a characteristic of $n \log n$ scaling. The error bars show the minimum and maximum execution times obtained during the testing process.

4.2 Test 2: Memory limitation

For this test batch, the program has been fed input from five different input files with varying memory limitations, ten times each. All of these files had 2^{23} attacks. The results can be seen in the chart below:

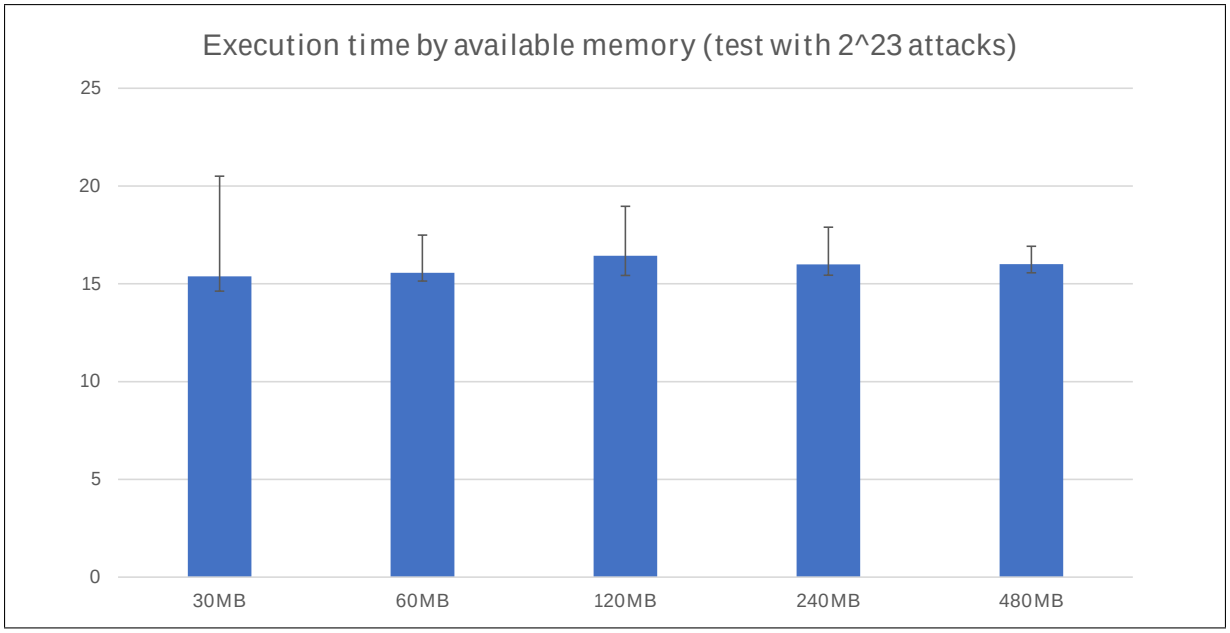

Figure 4: Chart of the execution time of the tests (in seconds) by the amount of memory available, in megabytes.

This test yielded very unusual results. Initially, one would expect that the execution time would become smaller and smaller as more memory became available to the sorting procedures. However, this wasn't the case - the program ran at about the same speed, regardless of how much memory was available. This can be explained by the expression that determines the number of times the sorting operation passes by the entire input file (or the equivalent amount of entries in the intermediate files):

$$\log_{\#ways}\left\lceil \frac{n}{2m} \right\rceil + 1 \tag{4.1}$$

Given that $\#ways = 17$, $n = 2^{23}$ and that an element in the heap has the size of five integers (x, y, number of tanks, flag, insertion order tracker), which is equivalent to 20B (in modern architechtures), the average amount of runs can be determined. If the amount of runs with 30MB and 480MB are to be compared, the results obtained are:

$$\log_{17}\left\lceil \frac{2^{23}}{2*480*1024*1024*\frac{1}{20}} \right\rceil + 1 = 1 \tag{4.2}$$

$$\log_{17}\left\lceil \frac{2^{23}}{2*30*1024*1024*\frac{1}{20}} \right\rceil + 1 = 2 \tag{4.3}$$

The difference between one and two passes doesn't seem to be particularly significant. This can be explained by the fact that other passes are necessary outside of the sorting operation: one to parse the raw input from stdin (which is many times larger than the parsed unsorted file, due to noise and the inefficiency of storing attacks in text form) and other to copy the sorted file back to standard output. Another explanation for this phenomenon could be that using an SSD helped mitigate the impact of disk operations in the program's execution.

5 Conclusion

In this project, string parsing and external sorting problems under limited primary memory have been addressed in a fictitious World War II scenario. An automaton has been used to parse the incoming raw data and the entries have been sorted by replacement selection. The procedures utilized were assymptotically analyzed and the entire program was proven to run at $n \log \frac{n}{2m}$ time.